Java Chapter 4: Loops

Contents

Time required: 60 minutes

DRY

Don't **R**epeat **Y**ourself

Learning Outcomes

Students will be able to:

- Write different types of loops, such as for and while
- Select the correct type of loop based on a given problem
- Use loops to solve common problems, such as, finding sum, finding max, etc.
- Write nested loops

Read: Think Java 2

- Chapter 6 Loops and Strings

Do: Java Tutorials

- Java While Loop
- Java For Loop
- Java For-Each Loop
- Java Break and Continue

Increment and Decrement Operators

For loops commonly use increment and decrement operators to help count through a loop.

Pre increment

```
x = 1
// ++ happens first, then x value is assigned to y
y = ++x
y = 2
x = 2
```

Post increment is the most used operator in a for loop. It is used when we want the values to count up.

```
X = 1
// x value is assigned to y, then increment x++
y = x++
y = 1
x = 2
```

Pre decrement

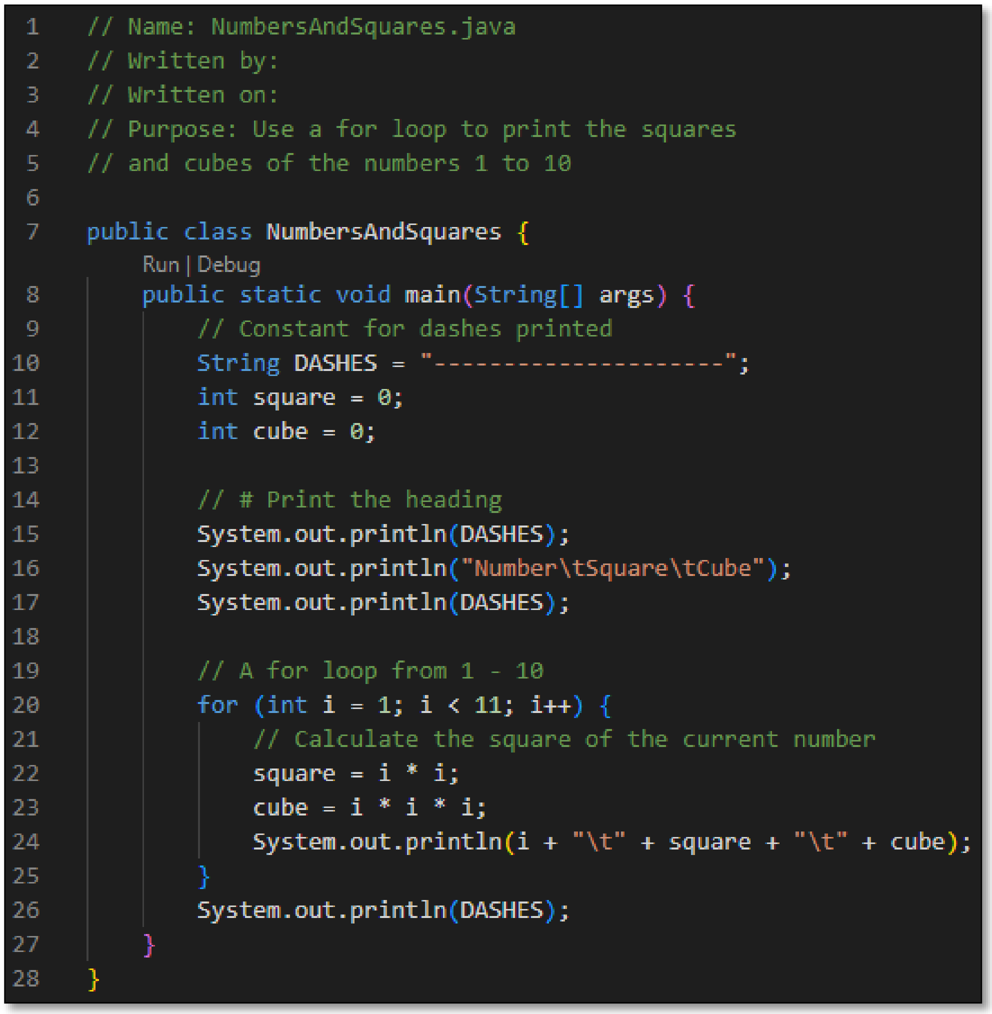
```
x = 1
// -- happens first, then x value is assigned to y
y = --x
y = 0
x = 0
```

Post decrement is the most used operator in a for loop. It is used when we want the values to count down.

```
X = 1
// x value is assigned to y, then decrement --
y = x--
y = 1
x = 0
```

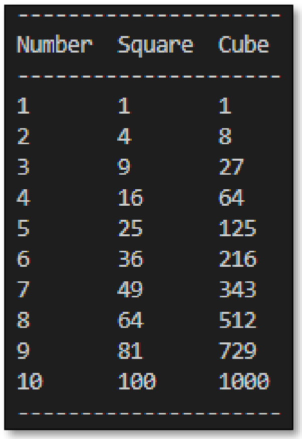
Tutorial 4.1 - For Loop Numbers and Squares

Create a Java program named: **NumbersAndSquares.java**



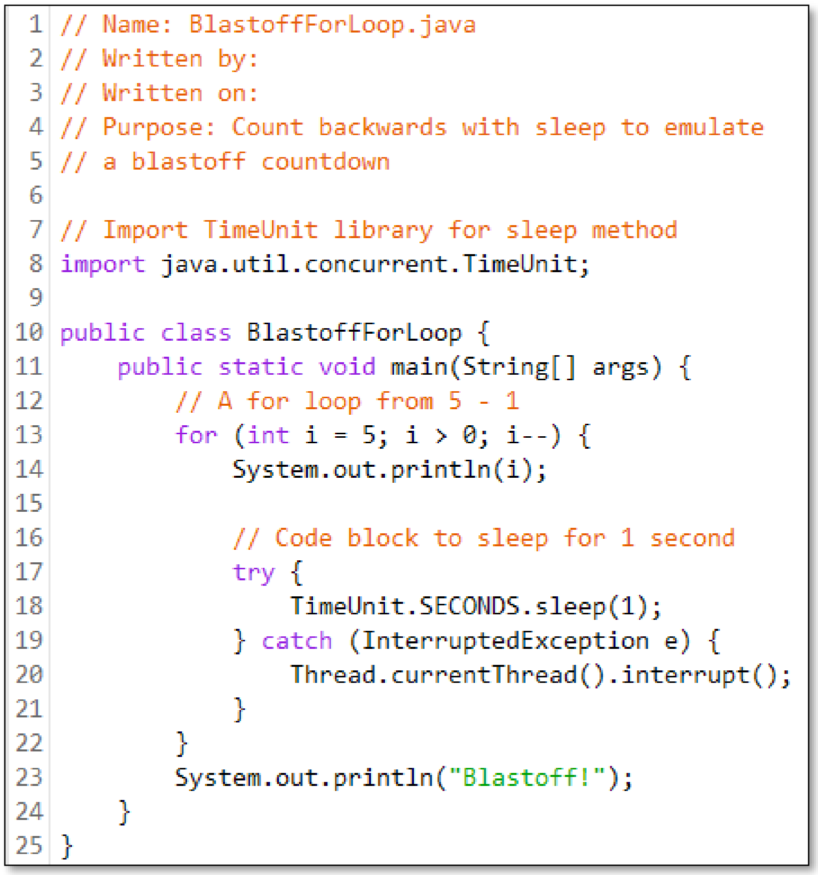
```java
// Name: NumbersAndSquares.java
// Written by:
// Written on:
// Purpose: Use a for loop to print the squares
// and cubes of the numbers 1 to 10

public class NumbersAndSquares {
    Run | Debug
    public static void main(String[] args) {
        // Constant for dashes printed
        String DASHES = "----------------------";
        int square = 0;
        int cube = 0;

        // # Print the heading
        System.out.println(DASHES);
        System.out.println("Number\tSquare\tCube");
        System.out.println(DASHES);

        // A for loop from 1 - 10
        for (int i = 1; i < 11; i++) {
            // Calculate the square of the current number
            square = i * i;
            cube = i * i * i;
            System.out.println(i + "\t" + square + "\t" + cube);
        }
        System.out.println(DASHES);
    }
}
```

Example run:

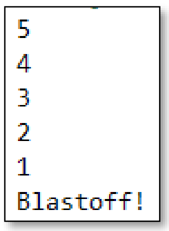

```
---------------------
Number  Square  Cube
---------------------
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
---------------------
```

Tutorial 4.2 - For Loop Blast Off!

Here is an example program that counts down from 5 and then prints a message. We imported the **sleep** method from the **TimeUnit** library, to count down 1 second at a time.


```java
// Name: BlastoffForLoop.java
// Written by:
// Written on:
// Purpose: Count backwards with sleep to emulate
// a blastoff countdown

// Import TimeUnit library for sleep method
import java.util.concurrent.TimeUnit;

public class BlastoffForLoop {
    public static void main(String[] args) {
        // A for loop from 5 - 1
        for (int i = 5; i > 0; i--) {
            System.out.println(i);

            // Code block to sleep for 1 second
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Blastoff!");
    }
}
```

Example program run:

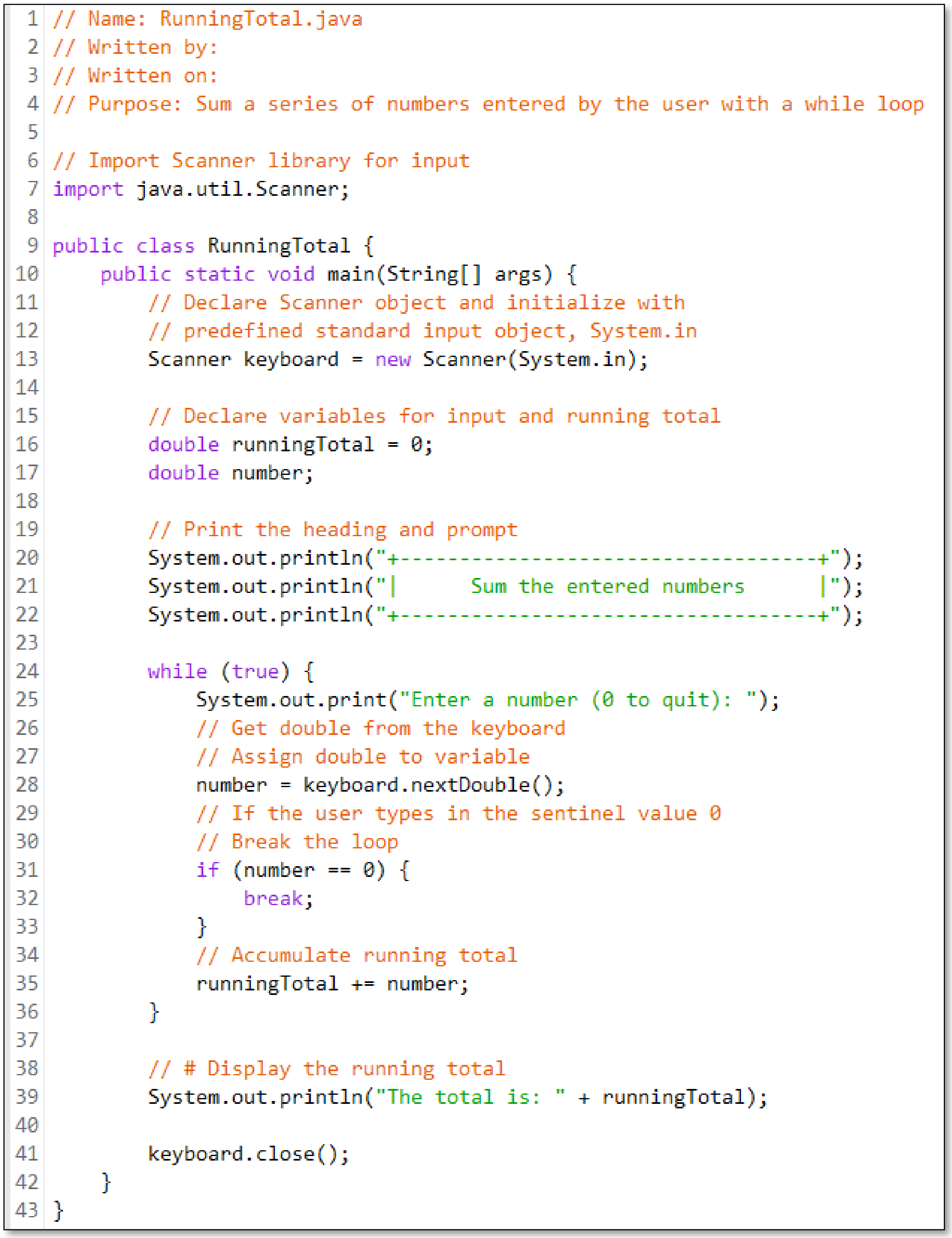


```
5
4
3
2
1
Blastoff!
```

Tutorial 4.3 - Running Total While Loop

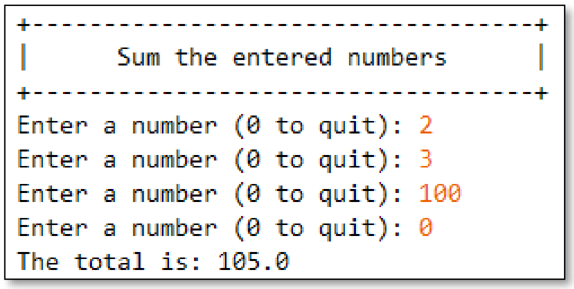
The following program demonstrates how to keep a running total of numbers entered by the user.

Create a Java program named: **RunningTotal.java**



```java
// Name: RunningTotal.java
// Written by:
// Written on:
// Purpose: Sum a series of numbers entered by the user with a while loop

// Import Scanner library for input
import java.util.Scanner;

public class RunningTotal {
    public static void main(String[] args) {
        // Declare Scanner object and initialize with
        // predefined standard input object, System.in
        Scanner keyboard = new Scanner(System.in);

        // Declare variables for input and running total
        double runningTotal = 0;
        double number;

        // Print the heading and prompt
        System.out.println("+-----------------------------------+");
        System.out.println("|      Sum the entered numbers      |");
        System.out.println("+-----------------------------------+");

        while (true) {
            System.out.print("Enter a number (0 to quit): ");
            // Get double from the keyboard
            // Assign double to variable
            number = keyboard.nextDouble();
            // If the user types in the sentinel value 0
            // Break the loop
            if (number == 0) {
                break;
            }
            // Accumulate running total
            runningTotal += number;
        }

        // # Display the running total
        System.out.println("The total is: " + runningTotal);

        keyboard.close();
    }
}
```

Example run:



```
+---------------------------------+
|      Sum the entered numbers      |
+---------------------------------+
Enter a number (0 to quit): 2
Enter a number (0 to quit): 3
Enter a number (0 to quit): 100
Enter a number (0 to quit): 0
The total is: 105.0
```

Input Validation and Exception Handling

Exception handling ensures that the correct data type is input by the user. If the user enters a string when an integer is what the program wants, this will generate a program exception.

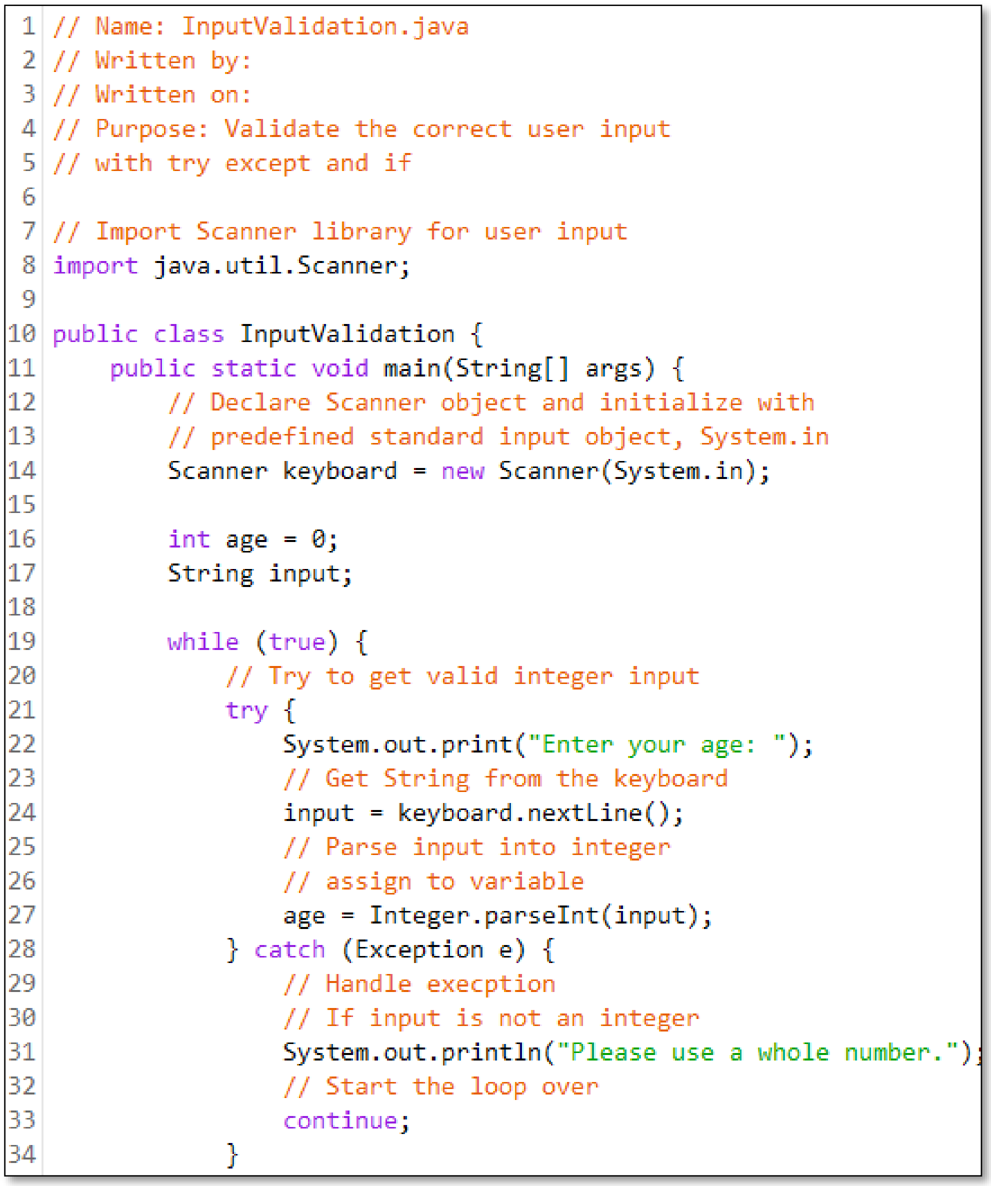
Input validation ensures that the input is in the right range. An age should be positive. A bank withdrawal would be negative.

For example, if you want users to enter their ages, your code shouldn't accept nonsensical answers such as negative numbers (which are outside the range of acceptable integers) or words (which are the wrong data type). Input validation can also prevent bugs or security vulnerabilities.

If you implement a **withdraw_from_account()** function that takes an argument for the amount to subtract from an account, you need to ensure the amount is a positive number. If the **withdraw_from_account()** function subtracts a negative number from the account, the "withdrawal" will end up adding money!

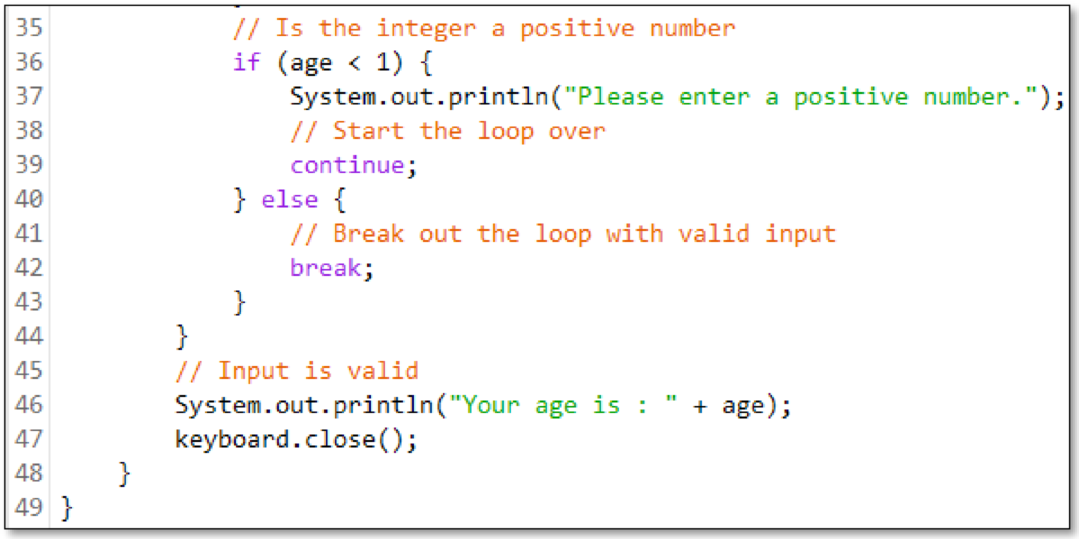
Typically, we perform input validation by repeatedly asking the user for input until they enter valid input, as in the following example:

Tutorial 4.4 - Input Validation Try Except While



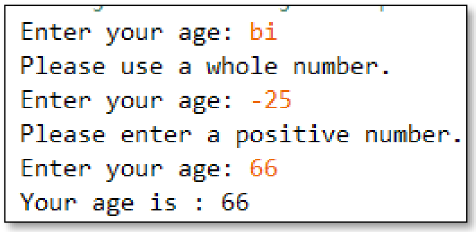
```java
// Name: InputValidation.java
// Written by:
// Written on:
// Purpose: Validate the correct user input
// with try except and if

// Import Scanner library for user input
import java.util.Scanner;

public class InputValidation {
    public static void main(String[] args) {
        // Declare Scanner object and initialize with
        // predefined standard input object, System.in
        Scanner keyboard = new Scanner(System.in);

        int age = 0;
        String input;

        while (true) {
            // Try to get valid integer input
            try {
                System.out.print("Enter your age: ");
                // Get String from the keyboard
                input = keyboard.nextLine();
                // Parse input into integer
                // assign to variable
                age = Integer.parseInt(input);
            } catch (Exception e) {
                // Handle execption
                // If input is not an integer
                System.out.println("Please use a whole number.");
                // Start the loop over
                continue;
            }
```

Sample run:



```java
            // Is the integer a positive number
            if (age < 1) {
                System.out.println("Please enter a positive number.");
                // Start the loop over
                continue;
            } else {
                // Break out the loop with valid input
                break;
            }
        }
        // Input is valid
        System.out.println("Your age is : " + age);
        keyboard.close();
    }
}
```

Example run:



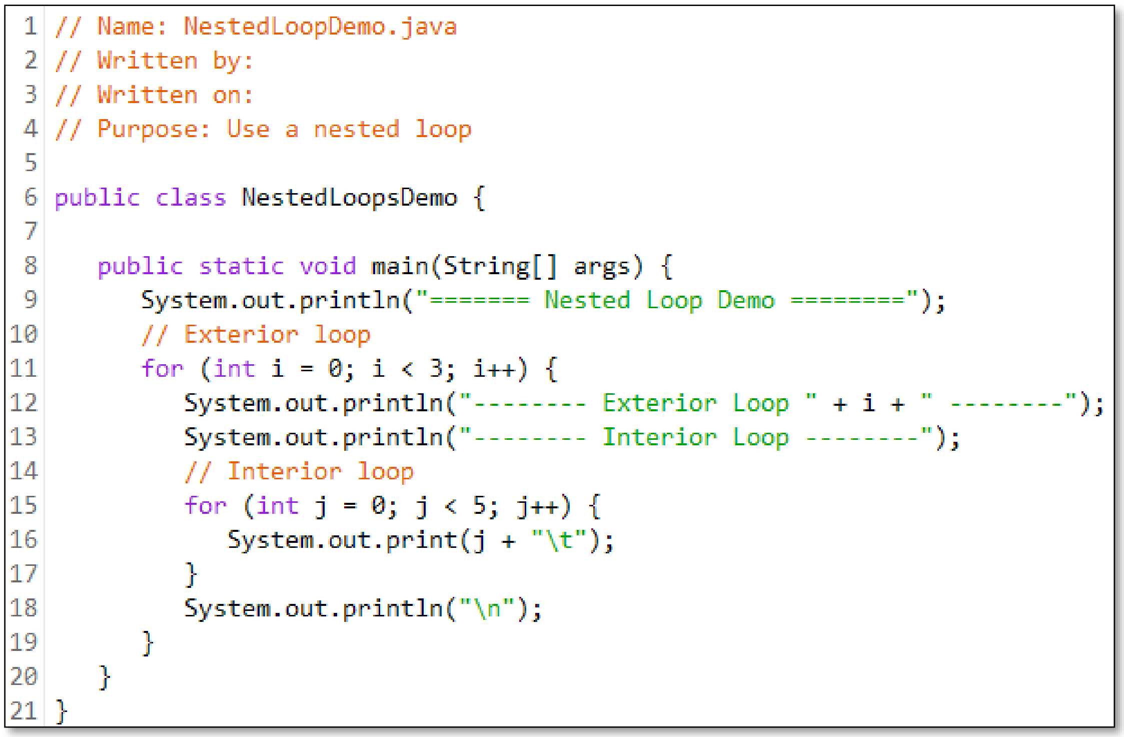
```
Enter your age: bi
Please use a whole number.
Enter your age: -25
Please enter a positive number.
Enter your age: 66
Your age is : 66
```

When you run this code, you'll be prompted for your age until you enter a valid one. This ensures that by the time the execution leaves the while loop, the age variable will contain a valid value that won't crash the program.

Tutorial 4.5 - Nested for Loop

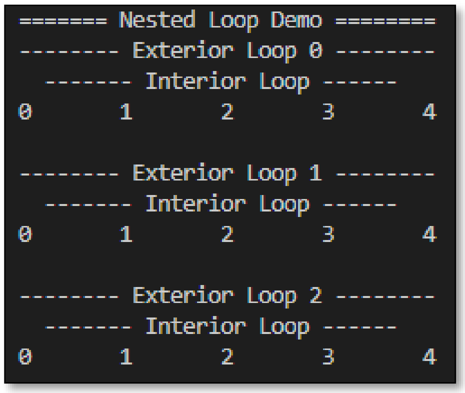
You can place a loop inside a loop.

Create a Java program named **NestedLoopsDemo.java**



```java
// Name: NestedLoopDemo.java
// Written by:
// Written on:
// Purpose: Use a nested loop

public class NestedLoopsDemo {

   public static void main(String[] args) {
      System.out.println("======= Nested Loop Demo ========");
      // Exterior loop
      for (int i = 0; i < 3; i++) {
         System.out.println("-------- Exterior Loop " + i + " --------");
         System.out.println("-------- Interior Loop --------");
         // Interior loop
         for (int j = 0; j < 5; j++) {
            System.out.print(j + "\t");
         }
         System.out.println("\n");
      }
   }
}
```

Example run:



```
======= Nested Loop Demo ========
-------- Exterior Loop 0 --------
  ------- Interior Loop ------
0       1       2       3       4

-------- Exterior Loop 1 --------
  ------- Interior Loop ------
0       1       2       3       4

-------- Exterior Loop 2 --------
  ------- Interior Loop ------
0       1       2       3       4
```

Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection.” For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you must search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.